Two papers on dynamic Huffman codes

J.S. Vitter

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 623

TWO PAPERS ON DYNAMIC HUFFMAN CODES

Jeffrey Scott VITTER

Février 1987

Two Papers on Dynamic Huffman Codes
Deux Articles sur les Codes Huffman Dynamiques

Jeffrey Scott Vitter[1,2]

Abstract. In the first paper we introduce and analyze a new one-pass algorithm for constructing dynamic Huffman codes and also analyze the one-pass algorithm due to Faller, Gallager, and Knuth. In each algorithm, both the sender and the receiver maintain equivalent dynamically varying Huffman trees, and the coding is done in real time. We show that the number of bits used by the new algorithm to encode a message containing t letters is $< t$ bits more than that used by the conventional two-pass Huffman scheme, independent of the alphabet size. This is best possible in the worst case, for any one-pass Huffman method. Tight upper and lower bounds are derived. Empirical tests show that the encodings produced by the new algorithm are shorter than those of the other one-pass algorithm and, except for long messages, are shorter than those of the two-pass method. It is well-suited for online encoding/decoding in data networks and for file compression. In the second paper, we present the details of a Pascal implementation of the new algorithm.

Résumé. Dans le premier article, nous introduissons et nous analysons un nouvel algorithme qui n'utilise qu'une seule passe pour construire une code Huffman dynamique. Nous considerons aussi l'algorithme de Faller, Gallager, and Knuth. Pour chaque algorithme, l'emetteur et le récepteur maintiennent les mêmes arbres de Huffman, qui changent dynamiquement en course de la transmission, et le coût de changement est moindre que le coût de transmission. Nous montrons que le nombre de bits utilisé par le nouvel algorithme pour coder un message de t éléments est moins que t bits en plus du nombre utilisé par la méthode bien connue de Huffman, qui fait deux passes. Cette situation est optimale, dans le « cas le pire », entre toutes les méthodes Huffman qui ne font qu'une passe. Nous derivons les limites précises inférieures et supérieures. Les experiences indiquent que les codes produits par le nouvel algorithme sont plus courts que pour l'algorithme qui ne fait qu'une passe, et qu'ils sont plus courts que pour la méthode bien connue quand le message n'est pas très long. Le nouvel algorithme a des applications dans les réseaux de données et pour la compression de fichers. Le deuxième article contient les détails d'une implementation en le langage Pascal.

---

[1] Support was provided in part by NSF research grant DCR–84–03613, by an NSF Presidential Young Investigator Award with matching funds from an IBM Faculty Development Award and an AT&T research grant, by an IBM research contract, and by a Guggenheim Fellowship.

[2] Current address: INRIA, Bâtiment 8, Domaine de Voluceau, Rocquencourt, B. P. 105, 78153 Le Chesnay Cedex, France. Part of this research was also done while the author was at Brown University in Providence, R. I., USA, at the Mathematical Sciences Research Institute in Berkeley, CA, USA, and at École Normale Supérieure in Paris. An earlier version of this manuscript appeared as Brown University Technical Report CS–85–13, and an extended abstract of the first paper appeared in *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, Portland, Oregon, USA, October 1985.

Design and Analysis of Dynamic Huffman Codes

1. Introduction

Variable-length source codes, such as those constructed by the well-known two-pass algorithm due to D. A. Huffman [1951], are becoming increasingly important for several reasons. Communication costs in distributed systems are beginning to dominate the costs for internal computation and storage. Variable-length codes often use fewer bits per source letter than do fixed-length codes such as ASCII and EBCDIC, which require $\lceil \log n \rceil$ bits per letter, where n is the alphabet size. This can yield tremendous savings in packet-based communication systems. Moreover, the buffering needed to support variable-length coding is becoming an inherent part of many systems.

The binary tree produced by Huffman's algorithm minimizes the weighted external path length $\sum_j w_j \ell_j$ among all binary trees, where w_j is the weight of the jth leaf, and ℓ_j is its depth in the tree. Let us suppose there are k distinct letters a_1, a_2, ..., a_k in a message to be encoded, and let us consider a Huffman tree with k leaves in which w_j, for $1 \leq j \leq k$, is the number of occurrences of a_j in the message. One way to encode the message is to assign a static code to each of the k distinct letters, and to replace each letter in the message by its corresponding code. Huffman's algorithm uses an optimum static code, in which each occurrence of a_j, for $1 \leq j \leq k$, is encoded by the ℓ_j bits specifying the path in the Huffman tree from the root to the jth leaf, where "0" means "to the left" and "1" means "to the right."

One disadvantage of Huffman's method is that it makes two passes over the data: one pass to collect frequency counts of the letters in the message, followed by the construction of a Huffman tree and transmission of the tree to the receiver; and a second pass to encode and transmit the letters themselves, based upon the static tree structure. This causes delay when used for network communication, and in file compression applications the extra disk accesses can slow down the algorithm. Faller [1973] and Gallager [1978] independently proposed a one-pass scheme, later improved substantially by Knuth [1985], for constructing dynamic Huffman codes. The binary tree that the sender uses to encode the $(t+1)$st letter in the message (and that the receiver uses to reconstruct the $(t+1)$st letter) is a Huffman tree for the first t letters of the message. Both sender and receiver start with the same initial tree and thereafter stay synchronized; they use the same algorithm to modify the tree after each letter is processed. Thus there is never need for the sender to transmit the tree to the receiver, unlike the case of the two-pass method. The processing time required to encode and decode a letter is proportional to the length of the letter's encoding, so the processing can be done in real time.

Of course, one-pass methods are not very interesting if the number of bits transmitted is significantly greater than with Huffman's two-pass method. This paper gives the first analytical study of the efficiency of dynamic Huffman codes. We derive a precise and clean characterization of the difference in length between the encoded message produced by a dynamic Huffman code and the encoding of the same message produced by a static Huffman code. The length (in bits) of the encoding produced by the algorithm of Faller, Gallager, and Knuth (Algorithm FGK) is shown to be

at most $\approx 2S + t$, where S is the length of the encoding by a static Huffman code, and t is the number of letters in the original message. More importantly, the insights we gain from the analysis lead us to develop a new one-pass scheme, which we call Algorithm Λ, that produces encodings of $< S + t$ bits. That is, compared with the two-pass method, Algorithm Λ uses less than one extra bit per letter. We prove this is optimum in the worst case among all one-pass Huffman schemes.

It is impossible to show that a given dynamic code is optimum among all dynamic codes, because one can easily imagine non-Huffman-like codes that are optimized for specific messages. Thus there can be no global optimum. For that reason we restrict our model of one-pass schemes to the important class of one-pass Huffman schemes, in which the next letter of the message is encoded based on a Huffman tree for the previous letters. We also do not consider the worst-case encoding length, among all possible messages of the same length, because for any one-pass scheme and any alphabet size n we can construct a message that is encoded with an average of $\geq \lfloor \log_2 n \rfloor$ bits per letter. The harder and more important measure, which we address in this paper, is the worst-case *difference in length* between the dynamic and static encodings of the same message.

One intuition why the dynamic code produced by Algorithm Λ is optimum in our model is that the tree it uses to process the $(t+1)$st letter not only is a Huffman tree with respect to the first t letters (that is, $\sum_j w_j \ell_j$ is minimized), but it also minimizes the external path length $\sum_j \ell_j$ and the height $\max_j \{\ell_j\}$ among all Huffman trees. This helps guard against a lengthy encoding for the $(t+1)$st letter. Our implementation is based on an efficient data structure we call a *floating tree*. Algorithm Λ is well-suited for practical use and has several applications. Algorithm FGK is already used for file compression in the *compact* command available under the 4.2BSD UNIX operating system [Mc Master, 1984]. Most Huffman-like algorithms use roughly the same number of bits to encode a message when the message is long; the main distinguishing feature is the coding efficiency for short messages, where overhead is more apparent. Empirical tests show that Algorithm Λ uses fewer bits for short messages than do Huffman's algorithm and Algorithm FGK. Algorithm Λ can thus be used as a general-purpose coding scheme for network communication and as an efficient subroutine in word-based compaction algorithms.

In the next section we review the basic concepts of Huffman's two-pass algorithm and the one-pass Algorithm FGK. In Section 3 we develop the main techniques for our analysis and apply them to Algorithm FGK. In Section 4 we introduce Algorithm Λ and prove that it runs in real time and gives optimal encodings, in terms of our model defined above. In Section 5, we describe several experiments comparing dynamic and static codes. Our conclusions are listed in Section 6.

2. Huffman's Algorithm and Algorithm FGK

In this section we discuss Huffman's original algorithm and the one-pass Algorithm FGK due to Faller, Gallager, and Knuth. First let us define the notation we shall use throughout the paper.

Definition 2.1. We define

n = alphabet size;
a_j = jth letter in the alphabet;
t = number of letters in the message processed so far;
$M_t = a_{i_1}, a_{i_2}, \ldots, a_{i_t}$, the first t letters of the message;
k = number of distinct letters processed so far;
w_j = number of occurrences of a_j processed so far;
ℓ_j = distance from the root of the Huffman tree to a_j's leaf;

The constraints are $1 \le j, k \le n$ and $0 \le w_j \le t$.

In many applications, the final value of t is much greater than n. For example, a book written in English on a conventional typewriter might correspond to $t \approx 10^6$ and $n = 87$. The ASCII alphabet size is $n = 128$.

Huffman's two-pass algorithm operates by first computing the letter frequencies w_j in the entire message. A leaf node is created for each letter a_j that occurs in the message; the weight of a_j's leaf is its frequency w_j. The meat of the algorithm is the following procedure for processing the leaves and constructing a binary tree of minimum weighted external path length $\sum_j w_j \ell_j$:

```
Store the k leaves in a list L;
while L contains at least two nodes do
    begin
    Remove from L two nodes x and y of smallest weight;
    Create a new node p, and make p the parent of x and y;
    p's weight := x's weight + y's weight;
    Insert p into L
    end;
```

The node remaining in L at the end of the algorithm is the root of the desired binary tree. We call a tree that can be constructed in this way a "Huffman tree." It is easy to show by contradiction that its weighted external path length is minimum among all possible binary trees for the given leaves. In each iteration of the **while**-loop, there may be a choice of which two nodes of minimum weight to remove from L. Different choices may produce structurally different Huffman trees, but all possible Huffman trees will have the same weighted external path length.

In the second pass of Huffman's algorithm, the message is encoded using the Huffman tree constructed in pass 1. The first thing the sender transmits to the receiver is the shape of the Huffman tree and the correspondence between the leaves and the letters of the alphabet. This is followed by the encodings of the individual letters in the message. Each occurrence of a_j is encoded by the sequence of 0s and 1s

that specifies the path from the root of the tree to a_j's leaf, using the convention that "0" means "to the left" and "1" means "to the right."

To retrieve the original message, the receiver first reconstructs the Huffman tree, based on the shape and leaf information. Then the receiver navigates through the tree by starting at the root and following the path specified by the 0 and 1 bits until a leaf is reached. The letter corresponding to that leaf is output, and the navigation begins again at the root.

Codes like this which correspond in a natural way to a binary tree are called "prefix codes," since the code for one letter cannot be a proper prefix of the code for another letter. The number of bits transmitted is equal to the weighted external path length $\sum_j w_j \ell_j$ plus the number of bits needed to encode the shape of the tree and the labeling of the leaves. Huffman's algorithm produces a prefix code of minimum length, since $\sum_j w_j \ell_j$ is minimized.

The two main disadvantages of Huffman's algorithm are its two-pass nature and the overhead required to transmit the shape of the tree. In this paper we explore alternative one-pass methods, in which letters are encoded "on the fly." We do not use a static code based on a single binary tree, since we are not allowed an initial pass to determine the letter frequencies necessary for computing an optimal tree. Instead the coding is based on a dynamically-varying Huffman tree. That is, the tree used to process the $(t+1)$st letter is a Huffman tree with respect to M_t. The sender encodes the $(t+1)$st letter a_{i_t} in the message by the sequence of 0s and 1s that specifies the path from the root to a_{i_t}'s leaf. The receiver then recovers the original letter by the corresponding traversal of its copy of the tree. Both sender and receiver then modify their copies of the tree before the next letter is processed so that it becomes a Huffman tree for M_{t+1}. The key point is that neither the tree nor its modification needs to be transmitted, because the sender and receiver use the same modification algorithm and thus always have equivalent copies of the tree.

Another key concept behind dynamic Huffman codes is the elegant so-called *Sibling Property*: A binary tree with p leaves of nonnegative weight is a Huffman tree if and only if

1. The p leaves have nonnegative weights $w_1, \ldots, w_p$, and the weight of each internal node is the sum of the weights of its children; and
2. The nodes can be numbered in nondecreasing order by weight, so that nodes $2j-1$ and $2j$ are siblings, for $1 \le j \le p-1$, and their common parent node is higher in the numbering.

The node numbering corresponds to the order in which the nodes are combined by Huffman's algorithm: nodes 1 and 2 are combined first, nodes 3 and 4 are combined second, nodes 5 and 6 are combined next, and so on.

Suppose that $M_t = a_{i_1}, a_{i_2}, \ldots, a_{i_t}$ has already been processed. The next letter $a_{i_{t+1}}$ is encoded and decoded using a Huffman tree for M_t. The main difficulty is how to modify this tree quickly in order to get a Huffman tree for M_{t+1}. Let us consider the example in Figure 1, for the case $t = 32$, $a_{i_{t+1}}$ = "b". It is not good enough to simply increment by 1 the weights of $a_{i_{t+1}}$'s leaf and its ancestors, because the resulting tree will not be a Huffman tree, as it will violate the sibling property. The nodes will no longer be numbered in nondecreasing order by weight; node 4 will have weight 6, but node 5 will still have weight 5. Such a tree could therefore not be

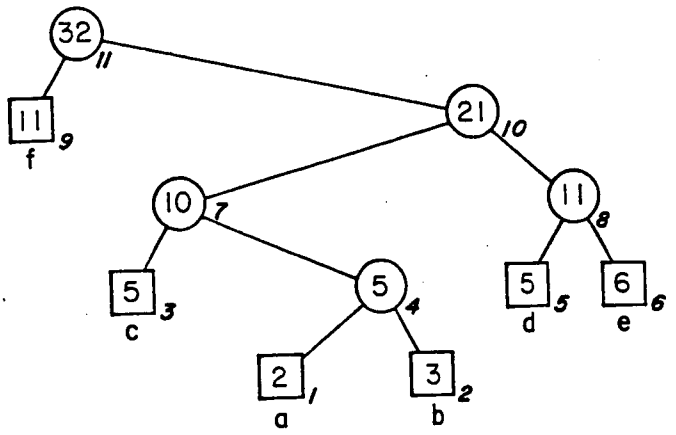
constructed via Huffman's two-pass algorithm.

The solution can most easily be described as a two-phase process (although for implementation purposes both phases can be combined easily into one). In the first phase, we transform the tree into another Huffman tree for M_t, to which the simple incrementing process described above can be applied successfully in phase 2 to get a Huffman tree for M_{t+1}. The first phase begins with the leaf of $a_{i_{t+1}}$ as the current node. We repeatedly interchange the contents of the current node, including the subtree rooted there, with that of the highest-numbered node of the same weight, and make the parent of the latter node the new current node. The current node in Figure 1(a) is initially node *2*. No interchange is possible, so its parent (node *4*) becomes the new current node. The contents of nodes *4* and *5* are then interchanged, and node *8* becomes the new current node. Finally, the contents of nodes *8* and *9* are interchanged, and node *11* becomes the new current node. The first phase halts when the root is reached. The resulting tree is pictured in Figure 1(b). It is easy to verify that it is a Huffman tree for M_t (that is, it satisfies the sibling property), since each interchange operates on nodes of the same weight. In the second phase, we turn this tree into the desired Huffman tree for M_{t+1} by incrementing the weights of $a_{i_{t+1}}$'s leaf and its ancestors by 1. Figure 1(c) depicts the final tree, in which the incrementing is done.

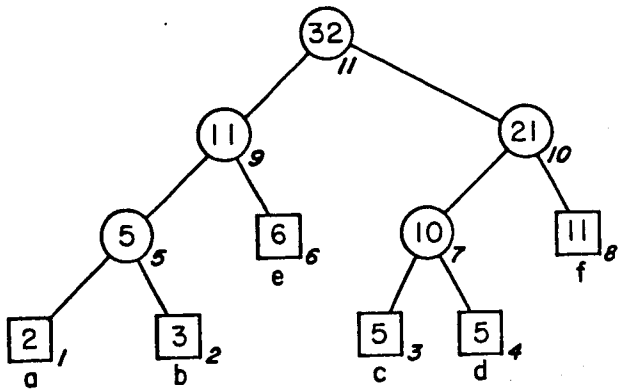
The reason why the final tree is a Huffman tree for M_{t+1} can be explained in terms of the sibling property: The numbering of the nodes is the same after the incrementing as before. Condition 1 and the second part of condition 2 of the sibling property are trivially preserved by the incrementing. We can thus restrict our attention to the nodes that are incremented. Before each such node is incremented, it is the largest-numbered node of its weight. Hence, its weight can be increased by 1 without becoming larger than that of the next node in the numbering, thus preserving the sibling property.

When $k < n$, we use a single 0-node to represent the $n - k$ unused letters in the alphabet. When the $(t + 1)$st letter in the message is processed, if it does not appear in M_t, the 0-node is split to create a leaf node for it, as illustrated in Figure 2. The $(t + 1)$st letter is encoded by the path in the tree from the root to the 0-node, followed by some extra bits that specify which of the $n - k$ unused letters it is, using a simple prefix code.

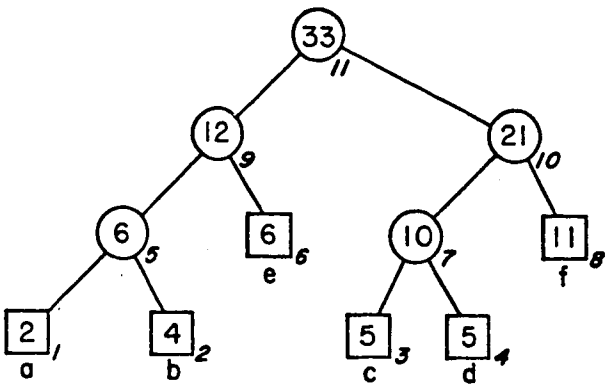
Phases 1 and 2 can be combined in a single traversal from the leaf of $a_{i_{t+1}}$ to the root, as shown below. Each iteration of the **while**-loop runs in constant time, with the appropriate data structure, so that the processing time is proportional to the encoding length. A full implementation appears in [Knuth, 1985].



(a)



(b)



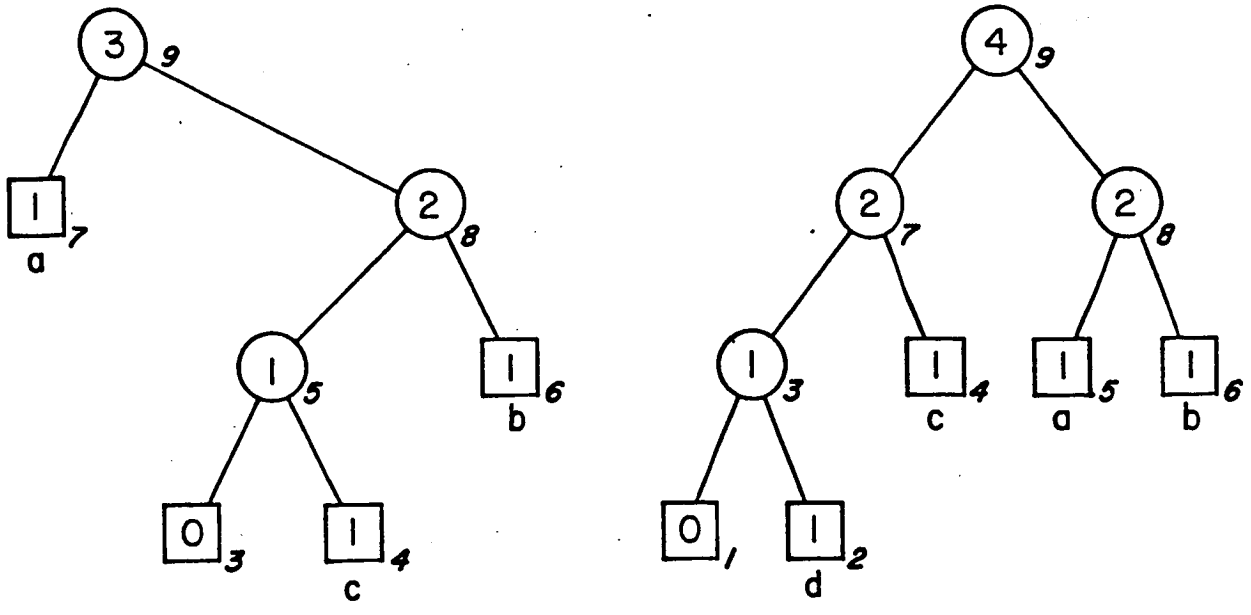
(c)

Figure 1. This example from [Knuth, 1985] for $t = 32$ illustrates the basic ideas of the Algorithm FGK. The node numbering for the sibling property is displayed next to each node. The next letter to be processed in the message is "$a_{i_{t+1}}$ = b". **(a)** The current status of the dynamic Huffman tree, which is a Huffman tree for M_t, the first t letters in the message. The encoding for "b" is "1011", given by the path from the root to the leaf for "b". **(b)** The tree resulting from the interchange process. It is a Huffman tree for M_t and has the property that the weights of the traversed nodes can be incremented by 1 without violating the sibling property. **(c)** The final tree, which is the tree in (b) with the incrementing done, is a Huffman tree for M_{t+1}.

```
procedure Update;
begin
q := leaf node corresponding to a_{i_{t+1}};
if (q is the 0-node) and (k < n − 1) then
    begin
    Replace q by a parent 0-node with two leaf 0-node children,
      numbered in the order left child, right child, parent;
    q := right child just created
    end;
if q is the sibling of a 0-node then
    begin
    Interchange q with the highest-numbered leaf of the same weight;
    Increment q's weight by 1;
    q := parent of q
    end;
while q is not the root of the Huffman tree do
    begin  {Main loop}
    Interchange q with the highest-numbered node of the same weight;
    {q is now the highest-numbered node of its weight}
    Increment q's weight by 1;
    q := parent of q
    end
end;
```

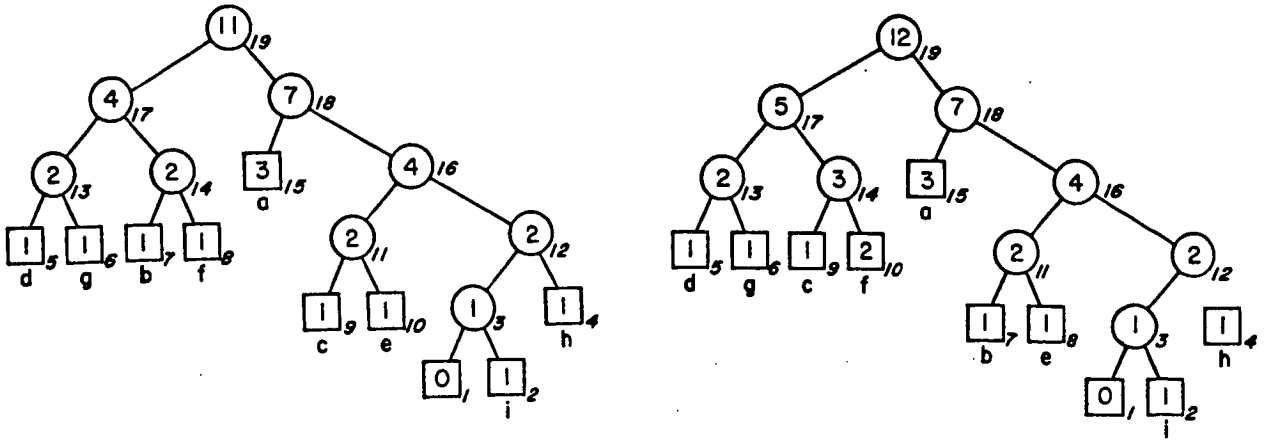
We shall denote an interchange in which q moves up one level by $\uparrow$ and an interchange between q and another node on the same level by $\rightarrow$. For example, in Figure 1, the interchange of nodes 8 and 9 is of type $\uparrow$, while that of nodes 4 and 5 is of type $\rightarrow$. Oddly enough, it is also possible for q to move down a level during an interchange, as illustrated in Figure 3; we shall denote such an interchange by $\downarrow$.

No two nodes with the same weight can be more than one level apart in the tree, except if one is the sibling of the 0-node. This follows by contradiction, since otherwise it will be possible to interchange nodes and get a binary tree having smaller external weighted path length. Figure 4 shows the result of what would happen if the letter "c" (rather than "d") were the next letter processed using the tree in Figure 2(a). The first interchange involves nodes two levels apart; the node moving up is the sibling of the 0-node. We shall designate this type of two-level interchange by $\uparrow\uparrow$. There can be at most one $\uparrow\uparrow$ per call to *Update*.



(a) (b)

Figure 2. Algorithm FGK operating on the message "abcd...". (a) The Huffman tree immediately before the fourth letter "d" is processed. The encoding for "d" is specified by the path to the 0-node, namely, "100". (b) After *Update* is called.



(a) (b)

Figure 3. (a) The Huffman tree formed by Algorithm FGK after processing "abcdefghiaa". (b) The Huffman tree that will result if the next processed letter is "f". Note that there is an interchange of type ↓ (between leaf nodes 8 and 10) followed immediately by an interchange of type ↑ (between internal nodes 11 and 14).

3. Analysis of Algorithm FGK

For purposes of comparing the coding efficiency of one-pass Huffman algorithms with that of the two-pass method, we shall count only the bits corresponding to the paths traversed in the trees during the coding. For the one-pass algorithms, we shall not count the bits used to distinguish which new letter is encoded when a letter is encountered in the message for the first time. And for the two-pass method, we shall not count the bits required to encode the shape of the tree and the labeling of the leaves. The noncounted quantity for the one-pass algorithms is typically between $k(\log_2 n - 1)$ and $k \log_2 n$ bits using a simple prefix code, and the uncounted quantity for the two-pass method is roughly $2k$ bits more than for the one-pass method. This means that our evaluation of one-pass algorithms will be *conservative* with respect to the two-pass method. When the message is long (that is, $t \gg n$), these uncounted quantities are insignificant compared with the total number of bits transmitted. (For completeness, the empirical results in Section 5 are include statistics that take into account these extra quantities.)

Definition 3.1. Suppose that a message $M_t = a_{i_1}, a_{i_2}, \ldots, a_{i_t}$ of size $t \geq 0$ has been processed so far. We define S_t to be the communication cost for a static Huffman encoding of M_t *using a Huffman tree based only on* M_t; that is,

$$S_t = \sum_j w_j \ell_j,$$

where the sum is taken over any Huffman tree for M_t. We also define s_t to be the "incremental" cost

$$s_t = S_t - S_{t-1}.$$

We denote by d_t the communication cost for encoding a_{i_t} using a dynamic Huffman code; that is,

$$d_t = \ell_{i_t},$$

for the dynamic Huffman tree with respect to M_{t-1}. We define D_t to be the total communication cost for all t letters; that is,

$$D_t = D_{t-1} + d_t, \qquad D_0 = 0.$$

Note that s_t does not have an intuitive meaning in terms of the length of the encoding for a_{i_t}, as does d_t. The following theorem bounds D_t by $\approx 2S_t + t$.

Theorem 3.1. *For each $t \geq 0$, the communication cost of Algorithm FGK can be bounded by*

$$S_t - k + \delta_{k=n} + \delta_{k<n} \min_{w_j > 0} \{w_j\} \leq D_t \leq 2S_t + t - 4k + 2\delta_{k=n} + 2\delta_{k<n} \min_{w_j > 0} \{w_j\} - m,$$

where δ_R denotes 1 if relation R is true and 0 otherwise, and m is the cardinality of the set $\{j \mid 1 \leq j \leq t;\ a_{i_j} \in M_{j-1}$; and $\forall x \in M_{j-1}$ such that $x \neq a_{i_j}$, x appears strictly more often in M_{j-1} than does $a_{i_j}\}$.

The term m is the number of times during the course of the algorithm that the processed leaf is not the 0-node and has strictly minimum weight among all other

leaves of positive weight. An immediate lower bound on m is $m \geq \min_{w_j>0}\{w_j\} - 1$. (For each value $2 \leq w \leq \min_{w_j>0}\{w_j\}$, consider the last leaf to attain weight w.) The minor δ terms arise because our one-pass algorithms use a 0-node when $k < n$, as opposed to the conventional two-pass method; this causes the leaf of minimum positive weight to be one level lower in the tree. The δ terms can be effectively ignored when there is a specially designated "end-of-file" character to denote the end of transmission, because when the algorithm terminates we have $\min_{w_j>0}\{w_j\} = 1$.

The Fibonacci-like tree in Figure 5 is an example where the first bound is tight. The difference $D_t - S_t$ decreases by 1 each time a letter not previously in the message is processed, except for when k increases from $n-1$ to n. The following two examples, in which the communication cost per letter D_t/t is bounded by a small constant, yield $D_t/S_t \to c > 1$. The message in the first example consists of any finite number of letters not including "a" and "b", followed by "abbaabbaa...". In the limit, we have $S_t/t \to 3/2$ and $D_t/t \to 2$, which yields $D_t/S_t \to 4/3 > 1$. The second example is a simple modification for the case of alphabet size $n = 3$. The message consists of the same pattern as above, without the optional prefix, yielding $D_t/S_t \to 2$. So far all the known examples where $\limsup_{t\to\infty} D_t/S_t \neq 1$ satisfy the constraint $D_t = O(t)$. We conjecture that the constraint is necessary:

Conjecture. *For each $t \geq 0$, the communication cost of Algorithm FGK satisfies*

$$D_t = S_t + O(t).$$

Before we can prove Theorem 3.1, we must develop the following useful notion. We shall denote by h_t the net change of height in the tree of the leaf for a_{i_t} as a result of interchanges during the tth call to *Update*.

Definition 3.2. For each $t \geq 1$, we define h_t by

$$h_t = (\#\ \uparrow\text{'s}) + 2(\#\ \uparrow\uparrow\text{'s}) - (\#\ \downarrow\text{'s}),$$

where we consider the interchanges that occur during the processing of the tth letter in the message.

The proof of Theorem 3.1 is based on the following important correspondence between h_t and $d_t - s_t$:

Theorem 3.2. *For $t \geq 1$, we have*

$$d_t - s_t = h_t - \delta_{\Delta k=1} + (\delta_{k<n \text{ or } \Delta k=1})\Delta \min_{w_j>0}\{w_j\},$$

where $\Delta f = (f \text{ after } a_{i_t} \text{ is processed}) - (f \text{ before } a_{i_t} \text{ is processed})$.

Proof. The δ terms are due to the presence of the 0-node when $k < n$. Let us consider the case in which there is no 0-node, as in Figure 1. We define $\mathcal{T}_a$ to be the Huffman tree with respect to M_{t-1}, $\mathcal{T}_b$ to be the Huffman tree formed by the interchanges applied to $\mathcal{T}_a$, and $\mathcal{T}_c$ to be the Huffman tree formed from $\mathcal{T}_b$ by incrementing a_{i_t}'s leaf and its ancestors. In the example in Figure 1, we have $t = 33$ and a_{i_t} = "b". The trees $\mathcal{T}_a$, $\mathcal{T}_b$, and $\mathcal{T}_c$ correspond to those in Figures 1(a), 1(b), and 1(c), respectively.

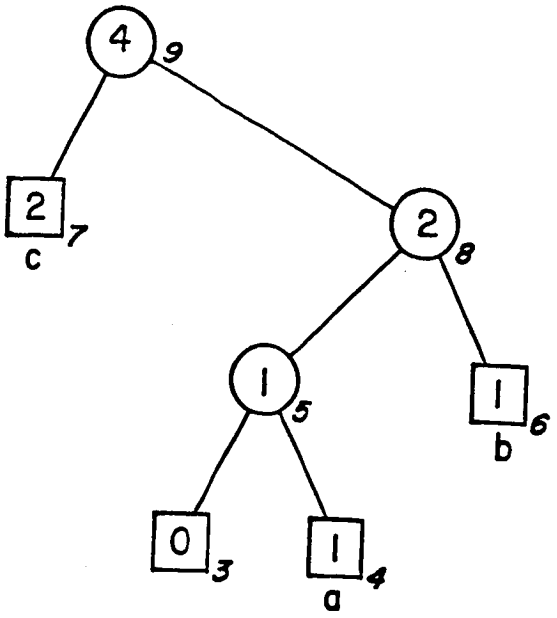


Figure 4. The Huffman tree that would result from Algorithm FGK if the fourth letter in the example in Figure 2 were "c" rather than "d". An interchange of type ↑↑ occurs when *Update* is called.

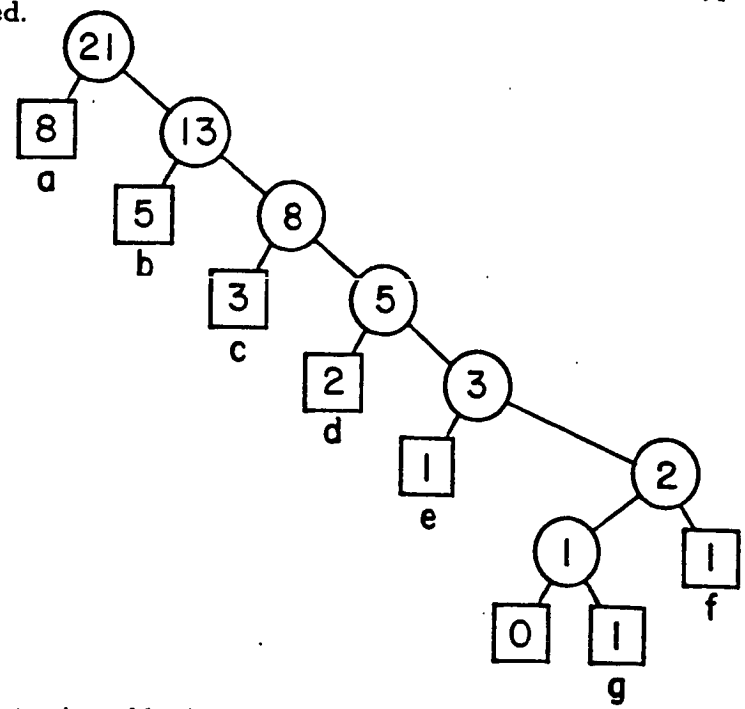


Figure 5. Illustration of both the lower bound of Theorem 3.1 and the upper bounds of Lemma 3.2. The sequence of letters in the message so far is "abacabdabaceabacabdf" followed by "g" and can be constructed via a simple Fibonacci-like recurrence. For the lower bound, let $t = 21$. The tree can be constructed without any exchanges of types ↑, ↑↑, or ↓; it meets the first bound given in Theorem 3.1. For the upper bound, let $t = 22$. The tree depicts the Huffman tree immediately before the tth letter is processed. If the tth letter is "h", we will have $d_t = 7$ and $h_t = \lceil d_t/2 \rceil - 1 = 3$. If instead the tth letter is "g", we will have $d_t = 7$ and $h_t = \lceil d_t/2 \rceil = 4$. If the tth letter is "f", we will have $d_t = 6$ and $h_t = \lfloor d_t/2 \rfloor = 3$.

Trees $\mathcal{T}_a$ and $\mathcal{T}_b$ represent Huffman trees with respect to $\mathcal{M}_{t-1}$, and $\mathcal{T}_c$ is a Huffman tree with respect to $\mathcal{M}_t$. The communication cost d_t for processing the tth letter a_{i_t} is the depth in $\mathcal{T}_a$ of its leaf node; that is,

$$d_t = \ell_{i_t}(\mathcal{T}_a). \tag{1}$$

Each interchange of type $\uparrow$ moves the leaf for a_{i_t} one level higher in the tree, each interchange of type $\uparrow\uparrow$ moves it two levels higher, and each interchange of type $\downarrow$ moves it one level lower. We have

$$h_t = \ell_{i_t}(\mathcal{T}_a) - \ell_{i_t}(\mathcal{T}_b). \tag{2}$$

The communication costs S_{t-1} and S_t are equal to the weighted external path lengths of $\mathcal{T}_a$ and $\mathcal{T}_c$, respectively. The interchanges that convert $\mathcal{T}_a$ to $\mathcal{T}_b$ maintain the sibling property, so $\mathcal{T}_a$ and $\mathcal{T}_b$ have the same weighted external path length. However, $\mathcal{T}_b$ is special since it can be turned into a Huffman tree for $\mathcal{M}_t$ (namely, tree $\mathcal{T}_c$) simply by incrementing a_{i_t}'s leaf and its ancestors by 1. Thus, we have

$$s_t = S_t - S_{t-1} = \ell_{i_t}(\mathcal{T}_b). \tag{3}$$

Putting (1), (2), and (3) together yields the result $d_t - s_t = h_t$.

When there is a 0-node present in $\mathcal{T}_a$, the communication cost S_{t-1} is $\min_{w_j(\mathcal{T}_a)>0}\{w_j(\mathcal{T}_a)\}$ less than the weighted external path length for $\mathcal{T}_a$, since the presence of the 0-node in $\mathcal{T}_a$ moves a leaf of minimum positive weight one level farther from the root than it would be if there were no 0-node. Similarly, when there is a 0-node in $\mathcal{T}_c$, the communication cost S_t is $\min_{w_j(\mathcal{T}_c)>0}\{w_j(\mathcal{T}_c)\}$ less than the weighted external path length for $\mathcal{T}_c$. If $\mathcal{T}_a$ has a 0-node and a_{i_t} appears in $\mathcal{M}_{t-1}$, then the 0-node will also appear in $\mathcal{T}_c$; this contributes a $\delta_{k<n}\Delta \min_{w_j>0}\{w_j\}$ term to $d_t - s_t$. If $\mathcal{T}_a$ has a 0-node and at most $n-2$ leaves of positive weight, and if a_{i_t} does not appear in $\mathcal{M}_{t-1}$, then the 0-node will be split, as outlined in *Update*; this has the effect of moving the leaf of a_{i_t} one level down, thus contributing $-(\delta_{k<n \text{ and } \Delta k=1})$ to $d_t - s_t$. The final special case is when the 0-node appears in $\mathcal{T}_a$ but not in $\mathcal{T}_c$; in this case, a_{i_t} does not appear in $\mathcal{M}_{t-1}$, but the other $n-1$ letters in the alphabet do. This contributes a $(\delta_{k=n \text{ and } \Delta k=1})(\Delta \min_{w_j>0}\{w_j\} - 1)$ term to $d_t - s_t$. Putting these three δ terms together gives us the final result. This proves Theorem 3.2. ∎

Three more lemmas are needed for the proof of Theorem 3.1:

Lemma 3.1. *During a call to Update in Algorithm FGK, each interchange of type $\downarrow$ is followed at some point by an $\uparrow$, with no $\downarrow$'s in between.*

Proof. By contradiction. Suppose that during a call to *Update* there are two interchanges of type $\downarrow$ with no $\uparrow$ in between. In the initial version of the tree before *Update* is called, let a_1 and b_1 be the nodes involved in the first $\downarrow$ interchange mentioned above, and let a_2 and b_2 be the nodes in the subsequent $\downarrow$ interchange; nodes a_1 and a_2 are one level higher in the tree than b_1 and b_2, respectively, before *Update* is called. Let $k \geq 1$ be the number of levels in the tree separating b_1 and a_2, and let a_1^k be the ancestor of a_1 that is k levels above a_1. Node a_1^k is one level higher than a_2, and we can show that their weights are equal by the following argument: If the

weight of a_1^k is $< a_2$'s weight, then we can interchange the two nodes and decrease the weighted external path length. On the other hand, if the weight of a_1^k is $> a_2$'s weight, then it can be shown from the sibling property that at some earlier point an interchange of type 1 should have occured, in which one of a_1's ancestors should have moved down one level. Both cases cause a contradiction. The $\downarrow$ interchange between a_2 and b_2 means that b_2 has the same weight as a_2 and is one level lower, which makes b_2 two levels lower than a_1^k but with the same weight. This is impossible, as mentioned at the end of Section 2. ∎

Lemma 3.2. *For each $t \geq 1$, we have*

$$0 \leq h_t \leq \begin{cases} \left\lceil \frac{d_t}{2} \right\rceil - 1, & \text{if } a_{i_t}\text{'s node is the 0-node;} \\ \left\lceil \frac{d_t}{2} \right\rceil, & \text{if } a_{i_t}\text{'s node is the 0-node's sibling;} \\ \left\lfloor \frac{d_t}{2} \right\rfloor, & \text{otherwise.} \end{cases}$$

An example achieving each of the three bounds is the Fibonacci-like tree given in Figure 5.

Proof. Let us consider what can happen when *Update* is called to process the tth letter a_{i_t}. Suppose for the moment that only interchanges of types $\uparrow$ or $\rightarrow$ occur. Each $\uparrow$ interchange, followed by the statement "$q :=$ parent of q", moves q two levels up in the tree. A $\rightarrow$ interchange or no interchange at all, followed by "$q :=$ parent of q", moves q up one level. Interchanges of type $\uparrow$ are not possible when q is a child of the root. Putting this all together, we find that the number of $\uparrow$ interchanges is at most $\lfloor d_t/2 \rfloor$, where d_t is the initial depth in the tree of the leaf for a_{i_t}.

If there are no interchanges of type $\uparrow\uparrow$ or $\downarrow$, the above argument yields $0 \leq h_t \leq \lfloor d_t/2 \rfloor$. If an interchange of type $\downarrow$ occurs, then by Lemma 3.1 there is a subsequent $\uparrow$, so the result still holds. An interchange of type $\uparrow\uparrow$ can occur if the leaf for a_t is the sibling of the 0-node; since at most one $\uparrow\uparrow$ can occur, we have $0 \leq h_t \leq \lceil d_t/2 \rceil$. The final case to consider occurs when the leaf for a_t is the 0-node; no interchange can occur during the first trip through the **while**-loop in *Update*, so we have $0 \leq h_t \leq \lceil d_t/2 \rceil - 1$. ∎

Lemma 3.3. *Suppose that a_{i_t} occurs in M_{t-1}, but strictly less often than all the other letters that appear in M_{t-1}. Then when the tth letter in the message is processed by Update, the leaf for a_{i_t} is not involved in an interchange.*

Proof. By the hypothesis, all the leaves other than the 0-node have a strictly larger weight than a_{i_j}'s leaf. The only node that can have the same weight is its parent. This happens when a_{i_t}'s leaf is the sibling of the 0-node, but there is no interchange in this special case. ∎

Proof of Theorem 3.1. By Lemma 3.2, we have $0 \leq h_t \leq d_t/2 + 1/2 - \delta_{\Delta k=1}$. Lemma 3.3 says that there are m values of t for which this bound can be lessened by 1. We get the final result by substituting this into the formula in Theorem 3.2 and by summing on t. This completes the proof. ∎

There are other interesting identies as well, besides the ones given above. For example, a proof similar to the one for Lemma 3.1 gives the following result:

Lemma 3.4. *In the execution of Update, if an interchange of type* $\uparrow$ *or* $\uparrow\uparrow$ *moves node* v *upward in the tree, interchanging it with node* x*, there cannot subsequently be more* $\uparrow$*'s than* $\downarrow$*'s until* q *reaches the lowest common ancestor of* v *and* x*.*

A slightly weaker bound of the form $D_t = 2S_t + O(t)$ can be proven using the following entropy argument suggested by B. Chazelle. The depth of a_{i_j}'s leaf in the dynamic Huffman tree during any of the w_j times a_{i_j} is processed can be bounded as a function of the leaf's relative weight at the time, which in turn can be bounded in terms of a_{i_j}'s final relative weight w_{i_j}/t. For example, during the last $\lfloor w_{i_j}/2 \rfloor$ times a_{i_j} is processed, its relative weight is $\geq w_{i_j}/(2t)$. The factor of 2 in front of the S_t term emerges because the relative weight of a leaf node in a Huffman tree can only specify the depth of the node to within a factor of 2 asymptotically (cf. Lemma 3.2). The characterization we give in Theorem 3.2 is robust in that it allows us to study precisely how $D_t - S_t$ changes as more letters are processed; this will be crucial for obtaining our main result in the next section that Algorithm Λ uses less than one extra bit per letter compared with the two-pass method.

4. Optimum Dynamic Huffman Codes

In this section, we describe Algorithm Λ and show that it runs in real time and is optimum in our model of one-pass Huffman algorithms. There were two motivating factors in its design:

1. The number of $\uparrow$'s should be bounded by some small number (in our case, 1) during each call to *Update*.
2. The dynamic Huffman tree should be constructed to minimize not only $\sum_j w_j \ell_j$, but also $\sum_j \ell_j$ and $\max_j\{\ell_j\}$, which intuitively has the effect of preventing a lengthy encoding of the next letter in the message.

Implicit Numbering

One of the key ideas of Algorithm Λ is the use of a different numbering scheme for the nodes than that used by Algorithm FGK. We shall use an *implicit numbering*, in which the node numbering corresponds to the visual representation of the tree. That is, the nodes of the tree are numbered in increasing order by level; nodes on one level are numbered lower than the nodes on the next higher level. Nodes on the same level are numbered in increasing order from left to right. We shall discuss later in this section how to maintain the implicit numbering via a floating tree data structure.

The node numbering used by Algorithm FGK does not always correspond to the implicit numbering. For example, the numbering of the nodes in Figures 1–2 and Figure 4 does agree with the implicit numbering, while the numbering in Figure 3 is quite different. The odd situation in which an interchange of type $\downarrow$ occurs, such as in Figure 3, can no longer happen when the implicit numbering is used. The following lemma lists some useful side-effects of implicit numbering.

Lemma 4.1. *With the implicit numbering, interchanges of type ↓ cannot occur. Also, if the node that moves up in an interchange of type ↑ is an internal node, then the node that moves down must be a leaf.*

Proof. The first result is obvious from the definition of implicit numbering. Suppose that an interchange of type ↑ occurs between two internal nodes a and b, where a is the node that moves up one level. In the initial tree, since a and b are on different levels, it follows from the sibling property that both a and b must have two children each of exactly half their weight. During the previous execution of the **while**-loop in *Update*, q is set to a's right child, which is the highest-numbered node of its weight. But this contradicts the fact that b's children have the same weight and are numbered higher in the implicit numbering. ∎

Invariant

The key to minimizing $D_t - S_t$ is to make ↑'s impossible, except for the first iteration of the **while**-loop in *Update*. We can do that by maintaining the following invariant:

> *For each weight w, all leaves of weight w precede (in the implicit numbering) all internal nodes of weight w.* (⋆)

Any Huffman tree satisfying (⋆) also minimizes $\sum_j \ell_j$ and $\max_j\{\ell_j\}$; this can be proven using the results of [Schwartz, 1964]. We shall see later that (⋆) can be maintained by floating trees in real time (that is, in $O(d_t)$ time for the tth processed letter).

Lemma 4.2. *If the invariant (⋆) is maintained, then interchanges of type ↑↑ are impossible, and the only possible interchanges of type ↑ must involve the moving up of a leaf.*

Proof. We shall prove both assertions by contradiction. We remarked at the end of Section 2 that no two nodes of the same weight can be two or more levels apart in the tree, if we ignore the sibling of the 0-node. The effect of the invariant (⋆) is to allow consideration of the 0-node's sibling. Let us denote the sibling by p and its weight by w. Suppose that there is another node p' of weight w two levels higher in the tree. By the invariant, node p' must be an internal node, since it follows p's parent (which also has weight w) in the implicit numbering. Each child of p' has weight $< w$, but follows p in the implicit numbering, thus contradicting the sibling property. For the second assertion, suppose there is an interchange of type ↑ in which an internal node moves up one level. By Lemma 4.1, the node that moves down must be a leaf. But this violates the invariant, since the leaf initially follows the internal node in the implicit numbering. ∎

The main result of the paper is the following theorem, which shows for Algorithm Λ that $D_t - S_t < t$.

Theorem 4.1. *For Algorithm Λ, we have*

$$S_t - k + \delta_{k=n} + \delta_{k<n} \min_{w_j>0}\{w_j\} \leq D_t \leq S_t + t - 2k + \delta_{k=n} + \delta_{k<n} \min_{w_j>0}\{w_j\} - m.$$

The δ terms and the term $m \geq \min_{w_j>0}\{w_j\} - 1$ have the same interpretation as in Theorem 3.1.

Proof. By Lemma 4.2, we have $0 \leq h_t \leq \delta_{\Delta k=0}$. In addition, there are m values of t where the upper bound on h_t can be decreased from 1 to 0. The theorem follows by plugging this into the bound in Theorem 3.2 (which holds not only for Algorithm FGK, but also for Algorithm Λ) and by summing on t. ∎

It is important to note that *the version of Update given in Section 2 is never executed by Algorithm Lambda.* An entirely different *Update* procedure, which is given later in this section, is called to maintain invariant ($\star$). But for purposes of analysis, a hypothetical execution of the former version of *Update* does provide via Theorem 3.2 a precise characterization of $d_t - s_t$.

The lower bound for D_t in Theorem 4.1 is the same as in Theorem 3.1., and the same example shows that it is tight. We can show that the upper bound is tight by generalizing the $D_t/S_t \to 4/3$ and $D_t/S_t \to 2$ examples in Section 3. For simplicity, let us assume that $n = 2^j + 1$, for some $j \geq 1$. We construct the message in a "balanced" fashion, so that the weights of $n-1$ letters are within 1 of one another, and the other letter has zero weight. The message begins with $n-1$ letters in the alphabet, once each. After $\mathcal{M}_{n-1}$ is processed, the leaves of the Huffman tree will be on the same level, except for two leaves on the next-lower level, one of which is the 0-node. At each step, the next letter in the message is defined inductively to be the current sibling of the 0-node, so as to force d_t to be always one more than the "average" depth of the leaves. We have $S_t = jt$ and $D_t = S_t + t - 2n + 3$, which matches the upper bound in Theorem 4.1, since $k = n-1$ and $m = \min_{w_j>0}\{w_j\}-1 = \lfloor t/(n-1) \rfloor - 1$. Another example consists appending the nth letter of the alphabet to the above message. In this case, we get $S_t = jt + \lfloor (t-1)/(n-1) \rfloor + 1$ and $D_t = S_t + t - 2n + 2 - \lfloor (t-1)/(n-1) \rfloor$, which again matches the upper bound, since $k = n$ and $m = \min_{w_j>0}\{w_j\} - 1 = \lfloor (t-1)/(n-1) \rfloor - 1$.

It is important to note that this construction works for any dynamic Huffman code. The corresponding value of D_t will be at least as large as the value of D_t for Algorithm Λ. This proves that Algorithm Λ is optimum in terms of the model we defined in Section 1:

Theorem 4.2. *Algorithm Λ minimizes the worst-case difference $D_t - S_t$, over all messages of length t, among all one-pass Huffman algorithms.*

Outline of Algorithm Λ

In order to maintain the invariant ($\star$), we must keep separate blocks for internal and leaf nodes.

Definition 4.1. *Blocks* are equivalence classes of nodes defined by $v \equiv x$ iff nodes v and x have the same weight and are either both internal nodes or both leaves. The *leader* of a block is the highest-numbered (in the implicit numbering) node in a block.

The blocks are linked together by increasing order of weight; a leaf block always precedes an internal block of the same weight. The main operation of the algorithm needed to maintain invariant ($\star$) is the *SlideAndIncrement* operation, illustrated in Figure 6. The version of *Update* we use for Algorithm Λ is outlined below:

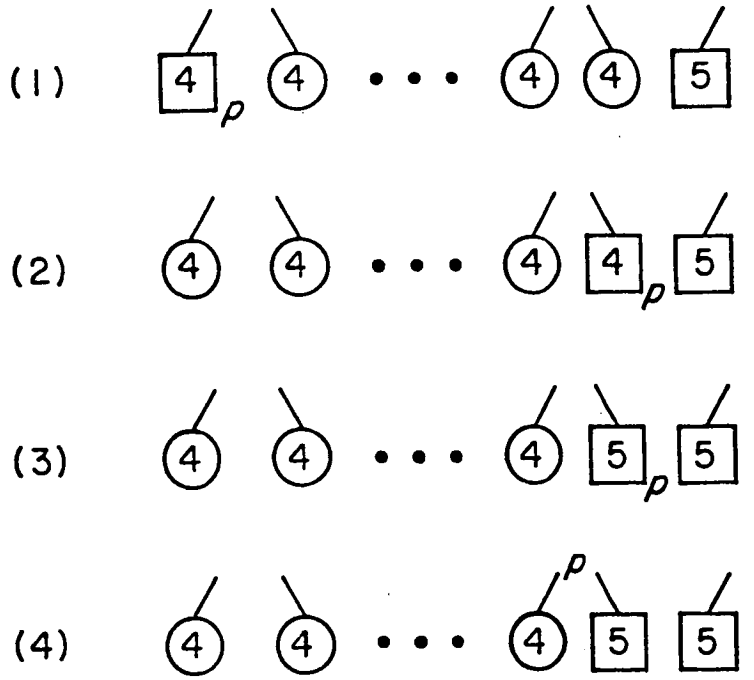
```
procedure Update;
begin
leafToIncrement := 0;
q := leaf node corresponding to a_{i_{t+1}};
if (q is the 0-node) and (k < n - 1) then
     begin  { Special Case # 1 }
     Replace q by an internal 0-node with two leaf 0-node children,
       such that the right child corresponds to a_{i_{t+1}};
     q := internal 0-node just created;
     leafToIncrement := the right child of q
     end
else begin
     Interchange q in the tree with the leader of its block;
     if q is the sibling of the 0-node then
          begin  { Special Case # 2 }
          leafToIncrement := q;
          q := parent of q
          end
     end;
while q is not the root of the Huffman tree do
     { Main loop; q must be the leader of its block }
     SlideAndIncrement(q);
if leafToIncrement ≠ 0 then  { Handle the two special cases }
     SlideAndIncrement(leafToIncrement)
end;

procedure SlideAndIncrement(p);
begin
wt := weight of p;
b := block following p's block in the linked list;
if ((p is a leaf) and (b is the block of internal nodes of weight wt))
  or ((p is an internal node) and
                (b is the block of leaves of weight wt + 1)) then
     begin
     Slide p in the tree ahead of the nodes in b;
     p's weight := wt + 1;
     if p is a leaf then p := new parent of p
     else p := former parent of p
     end
end;
```

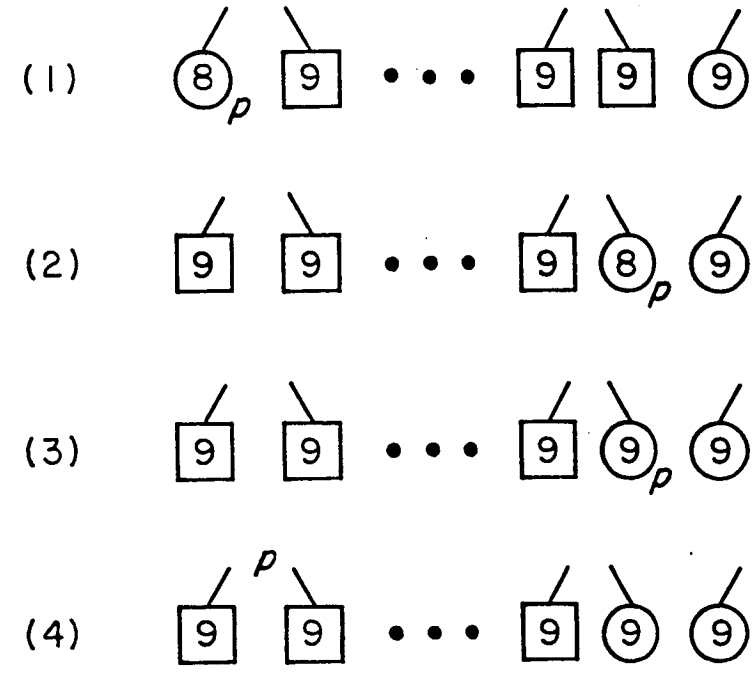
Examples of Algorithm A in operation are given in Figures 7–9; they depict the same examples used to illustrate Algorithm FGK in Figures 2, 4, and 5. As with Algorithm FGK, the processing can be done in $O(d_{t+1})$ time, if the appropriate data structure is used.

Data Structure

In this section we shall summarize the main features of our data structure for Algo-

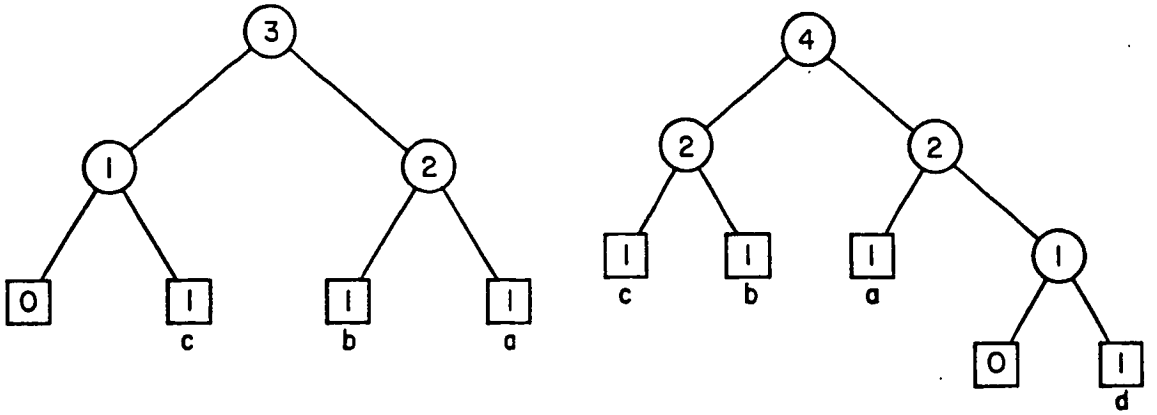


(a)



(b)

Figure 6. Algorithm A's *SlideAndIncrement* operation. All the nodes in a given block shift to the left one spot to make room for node p, which slides over the block to the right. **(a)** Node p is a leaf of weight 4. The internal nodes of weight 4 shift to the left. **(b)** Node p is an internal node of weight 8. The leaves of weight 9 shift to the left.



(a) (b)

Figure 7. Algorithm Λ operating on the message "abcd...". **(a)** The Huffman tree immediately before the fourth letter "d" is processed. **(b)** After *Update* is called.

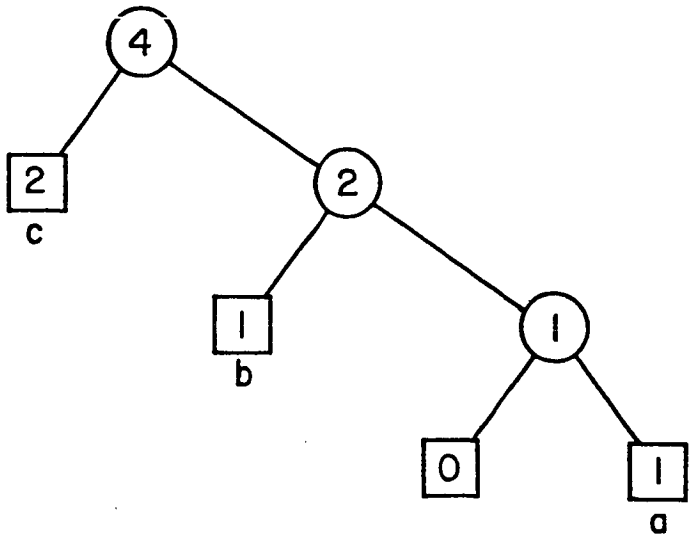


Figure 8. The Huffman tree that would result from Algorithm Λ if the fourth letter in the example in Figure 7 were "c" rather than "d".

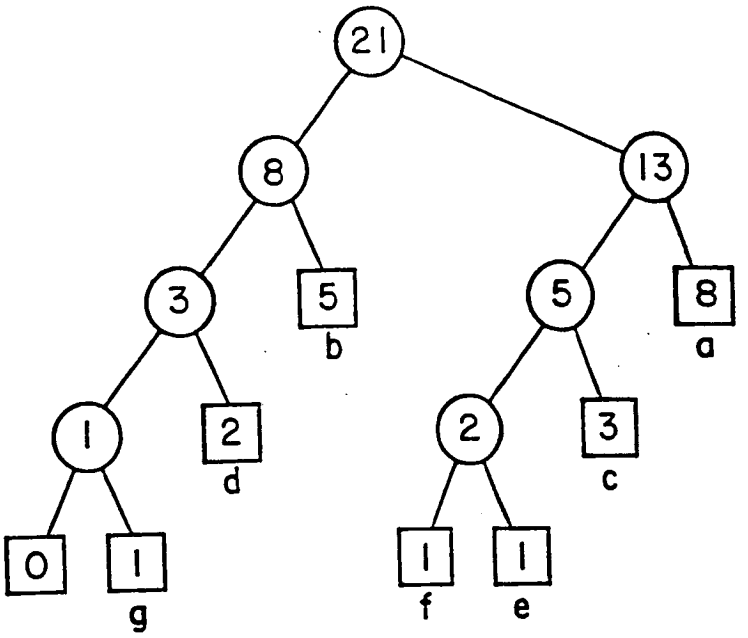


Figure 9. The Huffman tree constructed by Algorithm Λ for the same message used in Figure 5. Note how much shorter the tree is than in Figure 5.

rithm Λ. The details and implementation appear in the companion paper. The main operations that the data structure must support are:

- Represent a binary Huffman tree with nonnegative weights that maintains invariant (⋆).
- Store a contiguous list of internal tree nodes in nondecreasing order by weight; internal nodes of the same weight are ordered with respect to the implicit numbering. A similar list is stored for the leaves.
- Find the leader of a node's block, for any given node, based on the implicit numbering.
- Interchange the contents of two leaves of the same weight.
- Increment the weight of the leader of a block by 1, which can cause the node's implicit numbering to "slide" past the numberings of the nodes in the next block, causing their numberings to each decrease by 1.
- Represent the correspondence between the k letters of the alphabet that have appeared in the message and the positive-weight leaves in the tree.
- Represent the $n - k$ letters in the alphabet that have not yet appeared in the message by a single leaf 0-node in the Huffman tree.

The data structure makes use of an *explicit numbering*, which corresponds to the physical storage locations used to store information about the nodes. This is not to be confused with the implicit numbering defined in the last section. Leaf nodes are explicitly numbered n, $n - 1$, $n - 2$, ... in contiguous locations, and internal

nodes are explicitly numbered $2n-1$, $2n-2$, $2n-3$, ... contiguously; node q is a leaf iff $q \leq n$.

There is a close relationship between the explicit and implicit numberings, as specified in the second • listed above: For two internal nodes p and q, we have $p < q$ in the explicit numbering iff $p < q$ in the implicit numbering; the same holds for two leaves p and q.

The tree data structure is called a "floating tree" because the parent and child pointers for the nodes are not maintained explicitly. Instead, each block has a *parent* pointer and a *right_child* pointer that point to the parent and right child of the leader of the block. This allows a node to slide over an entire block without having to update more than a constant number of pointers. Because of the contiguous storage of leaves and of internal nodes, the locations of the parents and children of the other nodes in the block can be computed in constant time via an offset calculation from the block's *parent* and *right_child* pointer. Each execution of *SlideAndIncrement* thus takes constant time, so the encoding and decoding in Algorithm Λ can be done in real time.

The total amount of storage needed for the data structure is roughly $16n \log n + 15n + 2n \log t$ bits, which is about $4n \log n$ bits more than used by the implementation of Algorithm FGK in [Knuth, 1985]. The storage can be reduced slightly by extra programming. If storage is dynamically allocated, as opposed to preallocated via arrays, it will typically be much less. The running time is comparable to that of Algorithm FGK.

One nice feature of a floating tree, due to the use of implicit numbering, is that the parent of nodes $2j-1$ and $2j$ is less than the parent of nodes $2j+1$ and $2j+2$ in both the implicit and explicit numberings. Such an invariant is not maintained by the data structure in [Knuth, 1985]; see Figure 3(a), for example.

5. Empirical Results

We shall use S_t, D_t^{Λ} and D_t^{FGK} to denote the communication costs of Huffman's algorithm, Algorithms Λ, and Algorithm FGK. As pointed out at the beginning of Section 3, our evaluation of one-pass algorithms with respect to Huffman's two-pass method is conservative, since we are granting the two-pass method a handicap of $\approx 2k$ bits by not including in S_t the cost of representing the shape of the Huffman tree. The costs S_t, D_t^{Λ}, and D_t^{FGK} also do not count the bits required to encode the correspondence between the leaves of the tree and the letters of the alphabet that occur at least once in the message, but this can be expected to be about the same for the one-pass and two-pass schemes, roughly $k(\log_2 n - 1)$ to $k \log_2 n$ bits using a simple prefix code.

In this section we report on several experiments comparing the three algorithms in terms of coding efficiency. The tables below list not only the costs S_t, D_t^{Λ}, and D_t^{FGK}, but also the corresponding average number of bits used per letter of the message (denoted b/l for each of the three methods), which takes into account the bits needed to describe the tree and the labeling of the leaves. In terms of bits per letter b/l, *Algorithm Λ actually outperformed the two-pass method in all the*

experiments for which $t \leq 10^4$. Algorithm FGK used slightly more bits per letter, but also performed well.

Algorithm Λ has the advantage of using fewer bits per letter for small messages, where the differences in coding efficiency are relatively more significant. It can be shown using convergence theorems from statistics that, in the limit as $t \to \infty$, the communication cost of the one-pass Huffman algorithms is asymptotically equal to that of the two-pass method, for messages whose letters are generated independently according to some fixed probability distribution (discrete memoryless source). Even though the messages used in the longer of our experiments were not generated in such a manner, they are "sufficiently random" that it is not surprising that the statistics for the methods are very close for large t.

In the first experiment, the alphabet consisted of the 95 printable ascii characters, along with the end-of-line character, for a total of $n = 96$ letters. The message contained 960 letters: the 96 distinct characters repeated as a group 10 times. This is the type of example where all the methods can be expected to perform poorly. The static code does the worst. The results are summarized below at intervals of $t = 100$, 500, and 961:

t	k	S_t	b/l	D_t^{Λ}	b/l	D_t^{FGK}	b/l
100	96	664	13.1	569	10.2	659	11.2
500	96	3320	7.9	3225	7.4	3335	7.6
960	96	6400	7.1	6305	6.8	6415	6.9

The next example was a variation in which all the methods did very well. The message consisted of 10 repetions of the first character of the alphabet, followed by 10 repetitions of the second character, and so on, for a total message of $t = 960$ letters.

t	k	S_t	b/l	D_t^{Λ}	b/l	D_t^{FGK}	b/l
100	10	340	4.2	340	4.0	345	4.1
500	50	2860	6.5	2820	6.2	2863	6.3
960	96	6400	7.1	6305	6.8	6393	6.9

The third experiment was performed on the Pascal source code used to obtain the statistics for S_t and D_t^{Λ} reported in this section. Again, alphabet size $n = 96$ was used.

t	k	S_t	b/l	D_t^{Λ}	b/l	D_t^{FGK}	b/l
100	34	434	7.1	420	6.3	444	6.5
500	52	2429	5.7	2445	5.5	2489	5.6
1000	58	4864	5.3	4900	5.2	4953	5.3
10000	74	47710	4.8	47852	4.8	47938	4.8
12280	76	58457	4.8	58614	4.8	58708	4.8

The fourth experiment was run on the executable code compiled from the Pascal program mentioned above. The "letters" consisted of eight-bit characters in extended

ASCII, so alphabet size $n = 256$ was used.

t	k	S_t	b/l	D_t^{Λ}	b/l	D_t^{FGK}	b/l
100	9	124	2.1	117	1.9	122	2.0
500	9	524	1.2	517	1.2	522	1.2
1000	9	1024	1.1	1017	1.1	1022	1.1
10000	249	52407	5.5	52608	5.4	52868	5.5
34817	256	205688	6.0	206230	6.0	206585	6.0

The message for the final experiment consisted of the device-independent code for a technical book written in TEX [Vitter and Chen, 1987], with alphabet size $n = 256$.

t	k	S_t	b/l	D_t^{Λ}	b/l	D_t^{FGK}	b/l
100	40	372	7.7	345	6.7	378	7.0
500	123	2566	7.5	2514	6.9	2625	7.1
1000	177	5904	7.6	5875	7.2	6029	7.3
10000	248	67505	7.0	67769	6.9	67997	7.0
100000	256	691897	6.9	692591	6.9	692858	6.9
588868	256	4170298	7.1	4171314	7.1	4171616	7.1

6. Conclusions And Open Problems

The proposed Algorithm Λ performs real-time encoding and decoding of messages in a single pass, using less than one extra bit per letter to encode the message, as compared with Huffman's two-pass algorithm. This is optimum in the worst case, among all one-pass Huffman methods. The experiments reported in Section 5 indicate that the number of bits used by Algorithm Λ is roughly equal (and often better!) than that of the two-pass method. It has much potential for use in file compression and network communication and for hardware implementation.

Algorithm FGK peformed almost as well in the experiments. We conjecture that Algorithm FGK uses $O(1)$ extra bits per letter over the two-pass method in the worst case. Note that this does not contradict the examples that appear after Theorem 3.1, for which $D_t/S_t > 1$, since in each case the communication cost per letter is bounded, that is, $D_t = O(t)$. Figure 5 shows that $d_t \neq s_t + O(1)$, so a proof of the conjecture would require an amortized approach.

The one-pass Huffman algorithms we discuss in this paper can be generalized to d-way trees, for $d \geq 2$, for the case in which base-d digits are transmitted rather than bits. Algorithm Λ can also be modified to support the use of a "window" of size $b > 0$, as in [Knuth, 1985]. Whenever the next letter in the message is processed, its weight in the tree is increased by 1, and the weight of the letter processed b letters ago is decreased by 1. This technique would work well for the second experiment reported in the previous section.

Huffman coding does not have to be done letter by letter. An alternative well suited for file compression in some specific domains is to break up the message into maximal-length alphanumeric words and nonalphanumeric words. Each such word is treated as a single "letter" of the alphabet. One Huffman tree can be used for the alphanumeric words, and another for the nonalphanumeric words. The final sizes of the Huffman trees are proportional to the number of distinct words used. In many computer programs written in a high-level language, for example, the vocabulary consists of some variable names and a few frequently used keywords, such as "**while**", "**if**", and "**end**", so the alphabet size is reasonable. The alphabet size must be bounded beforehand in order for one-pass Huffman algorithms to work efficiently.

Algorithm Λ can also be used to enhance other compression schemes, such as the one-pass method described and analyzed in [Bentley, Sleator, Tarjan, and Wei, 1984], which is typically used in a word-based setting. A self-organizing cache of size c is used to store representatives of the last c distinct words encountered in the message. When the the tth word in the message is processed, if its representative is currently in position ℓ in the cache, then the word is encoded by an encoding of ℓ, using a suitable prefix code; otherwise, the word is encoded by an encoding of $c + 1$, and extra bits are used to specify which of the words not in the cache it is. Its representative is then moved to the front of the cache, bumping other representatives down by one if necessary, and the next word in the message is processed. Similar algorithms are also considered in [Elias, 87]. The algorithm can be made to run in real time by use of balanced tree techniques, and it uses no more than $S_t + t + 2t \log(1 + S_t/t)$ bits (cf. Theorem 4.1) to encode a message containing t words, not counting the extra bits required when the representative is not in the cache. (For any given word that appears more than once in the message, its representative can potentially be absent from the cache each time it is processed, and whenever it is absent, extra bits are required.) The method achieves its best coding efficiency when the prefix code used to encode the relative cache position ℓ is the dynamic Huffman code constructed by Algorithm Λ.

Acknowledgements. The author would like to thank Marc Brown, Bernard Chazelle, and Bob Sedgewick for interesting discussions. Marc's animated Macintosh implementation of Algorithm FGK helped greatly in the testing of Algorithm Λ and in the preparation of the figures. The entropy argument mentioned at the end of Section 3 is due to Bernard. Bob suggested the $D_t/S_t \to 4/3$ example in Section 3. Thanks also go to the referees for their very helpful comments.

References

J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. "A Locally Adaptive Data Compression Scheme," *Communications of the ACM, 29*, 4 (April 1986), 320–330.

P. Elias. "Interval and Recency-Rank Source Coding: Two Online Adaptive Variable-Length Schemes," *IEEE Transactions on Information Theory* (to appear).

N. Faller. "An Adaptive System for Data Compression," *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers* (1973), 593–597.

R. G. Gallager. "Variations on a Theme by Huffman," *IEEE Transactions on Information Theory, IT-24*, 6 (November 1978), 668–674.

D. A. Huffman. "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the Institute of Radio Engineers, 40* (1951), 1098–1101.

D. E. Knuth. "Dynamic Huffman Coding," *Journal of Algorithms, 6* (1985), 163–180.

E. S. Schwartz. "An Optimum Encoding with Minimum Longest Code and Total Number of Digits," *Information and Control, 7*, 1 (March 1964), 37–44.

C. L. Mc Master. "Documentation of the *Compact* Command," *UNIX User's Manual*, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, University of California (March 1984).

J. S. Vitter and W. C. Chen. *Design and Analysis of Coalesced Hashing.* Oxford University Press, New York (1987).

Implementation of Algorithm Λ

1. Introduction

In this paper we present a Pascal implementation of Algorithm Λ, the one-pass algorithm for constructing dynamic Huffman codes that is described and analyzed in the companion paper. The dynamic code used to process the $(t+1)$st letter in the message is a Huffman code based upon the first t letters. The processing time for each letter is proportional to the length of its encoding, so the program runs in real time. It is shown in the companion paper that the number of bits used by Algorithm Λ to encode a message of t letters is less than t bits more than that used by the well-known two-pass algorithm developed by D. A. Huffman [1951]. Algorithm Λ is optimum in this respect among all one-pass Huffman schemes. Experiments indicate that Algorithm Λ typically uses fewer bits than Huffman's algorithm and other one-pass Huffman methods. The algorithm has applications in file compression and network transmission.

We shall use the following terminology in our discussion:

n = alphabet size;
a_j = jth letter in the alphabet;
t = number of letters in the message processed so far;
k = number of distinct letters processed so far.

Each letter in the message is encoded by the path from the root to its leaf node in the current version of the Huffman tree. The weight of each leaf is the number of times the corresponding letter has appeared previously in the message. When $k < n$, a 0-node is used to represent the $n - k$ unused letters in the message. When a letter appears in the message for the first time, additional bits are put into the encoding to specify which of the unused letters it is, and if there are still unused letters remaining, the 0-node is split to create an extra leaf to accomodate the new letter.

One of the main features of Algorithm Λ is its use of *implicit numbering*, in which the nodes in the Huffman tree are numbered in increasing order from bottom to top and in each level in increasing order from left to right. Another main feature is the invariant that all leaves of a given weight precede in the implicit numbering all internal nodes of the same weight. These two features are shown in the companion paper to guarantee good coding efficiency.

In this paper we present a detailed implementation for Algorithm Λ. The data structure is described in the next section, and Section 3 contains the Pascal code.

2. Data Structure

We shall denote all the leaves of a given weight as a *leaf block* and all the internal nodes of a given weight as an *internal block*. The largest-numbered node in a block is called the *leader* of the block. Our invariant implies that the blocks are linked together in increasing order by weight, and that a leaf block always precedes an internal block of the same weight. The main operations that must be supported by the data structure for Algorithm Λ are:

- Represent a binary Huffman tree with nonnegative weights that maintains the invariant.
- Store a contiguous list of internal tree nodes in nondecreasing order by weight; internal nodes of the same weight are ordered with respect to the implicit numbering. A similar list is stored for leaf nodes.
- Find the leader of a node's block, for any given node, based upon the implicit numbering.
- Interchange the contents of two leaves of the same weight.
- Increment the weight of the leader of a block by 1, which can cause the node's implicit numbering to "slide" past the implicit numberings of the nodes in the next block, causing their implicit numberings to each decrease by 1.
- Represent the correspondence between the k letters of the alphabet that have appeared in the message and the positive-weight leaves in the tree.
- Represent the $n - k$ letters in the alphabet that have not yet appeared in the message by a single leaf 0-node in the Huffman tree.

The components of the data structure are listed below. The number of leaves of zero weight is specified by integer variables M, E, and R:

$$\begin{aligned} M = n - k &= \text{the number of zero-weight letters in the alphabet} \\ &= 2^E + R, \text{ where } 0 \le R < 2^E, \text{ except that } R = -1 \text{ when } M = 0. \end{aligned}$$

The data structure makes use of an *explicit numbering*, which corresponds to the physical storage locations used to store information about the nodes. This is not to be confused with the implicit numbering defined in the last section. Unless otherwise stated, all references to node numberings in this section are based upon the explicit numbering. Leaf nodes are explicitly numbered $1, \ldots, n$ in contiguous locations in physical memory, and internal nodes are explicitly numbered $n + \max\{1, M\}, \ldots, 2n - 1$ in contiguous locations in memory. Node q is a leaf node iff $q \le n$. When $k < n$ (that is, when $M > 0$), the 0-node in the Huffman tree is node M, and the positive-weight leaves are nodes $M + 1, \ldots, n$. Nodes $1, \ldots, M$ represent letters of zero weight, though only node M actually appears in the Huffman tree. When $k > 1$ (that is, when $M < n$), the root of the tree is internal node $2n - 1$; otherwise, we have $M = n$ and the root of the tree is node n, the 0-node.

There is a close relationship between the explicit and implicit numberings, as specified in the second • listed above: For two internal nodes p and q, we have $p < q$ in the explicit numbering iff $p < q$ in the implicit numbering; the same holds for two leaf nodes p and q.

The tree data structure is called a "floating tree" because the parent and child pointers for the nodes are not maintained explicitly. Instead, each block has a *parent*

pointer and a *rtChild* pointer that point to the parent and right child of the leader of the block. This allows a node to slide over an entire block without having to update more than a constant number of pointers. Because of the contiguous storage of leaves and of internal nodes, the locations of the parents and children of the other nodes in the block can be computed in constant time via an offset calculation from the block's *parent* and *rtChild* pointer.

The correspondence between leaf nodes and the letters they represent is given by the arrays *alpha* and *rep*:

$alpha[q] = j$, $1 \le j \le n$, iff a_j is the letter represented by node q, $1 \le q \le n$.

$rep[j] = q$, $1 \le q \le n$, iff node q corresponds to letter a_j, $1 \le j \le n$.

The main entity in the floating tree representation is the block, as defined earlier. Blocks are numbered in the range $1, \ldots, 2n-1$ in no particular order. The mapping between nodes and blocks is given by

$block[q]$ = block number of node q, for $\max\{1, M\} \le q \le n$ or $n+\max\{1, M\} \le q \le 2n-1$.

The following eight arrays of integers are each indexed by a block number b in the range $1 \le b \le 2n-1$:

$weight[b]$ = weight of each node in block b.

$parent[b]$ = the parent node of the leader of block b, if it exists; and 0 otherwise.

$parity[b]$ = 0 if the leader of block b is a left child or the root of the Huffman tree; and 1 otherwise.

$rtChild[b]$ = q if b is a block of internal nodes and node q is the right child of the leader of block b.

$first[b]$ = q if node q is the leader of block b.

$last[b]$ = q if node q is smallest-numbered node in block b.

$prevBlock[b]$ = previous block on the circularly-linked list of blocks.

$nextBlock[b]$ = next block on the circularly-linked list of blocks.

Each slot in the array *weight* must be capable of storing any integer in the range $[0, t]$. The unused blocks are linked together using *nextBlock* in a list headed by

availBlock = first block in the available-block list if the list is nonempty; and 0 otherwise.

The final component of the data structure is an array indexed by $1 \le i \le n$:

$stack[i]$ = ith-to-last bit of the encoding of the current letter being processed.

Except for the elements of the array *weight*, each integer variable can take on at most n or $2n-1$ values, which requires either $\lceil \log_2 n \rceil$ or $\lceil \log_2 n \rceil + 1$ bits of storage. The total amount of storage (in bits) needed for the data structure is

$$2\lceil \log_2 n \rceil + \lceil \log_2 \log_2 n \rceil + 2n\lceil \log_2 n \rceil + (2n-1)\,(\lceil \log_2 t \rceil + 7\lceil \log_2 n \rceil + 7) + \lceil \log_2 n \rceil + n$$
$$\approx 16n\lceil \log_2 n \rceil + 15n + 2n\lceil \log_2 t \rceil,$$

which is only about $4n\lceil \log_2 n \rceil$ more bits of storage than used by Algorithm FGK. The storage requirement can be reduced by roughly $n\lceil \log_2 n \rceil$ bits if separate available-block lists are kept for internal nodes and leaf nodes, since leaf blocks do not need a *rtChild* value. If storage is dynamically allocated, as opposed to preallocated via arrays, it will typically be much less.

3. Pascal Code

The basic implementation of Algorithm Λ is along the lines of the implementation of the one-pass algorithm in [Knuth, 85]. The primary difference between the two is that Algorithm Λ uses the implicit node numbering and the floating tree data structure in order to maintain the invariant defined in Section 1.

The basic loop for encoding and transmitting a message is

Initialize;
while there are more letters to encode **do begin**
    Let a_j be the next letter to encode;
    EncodeAndTransmit(j);
    Update(j)
    end;

The corresponding loop for receiving and decoding a message is

Initialize;
while there is more to decode **do begin**
    $j :=$ *ReceiveAndDecode*;
    Output(a_j);
    Update(j)
    end;

The *Initialize* procedure forms an initial Huffman tree consisting of a single leaf 0-node. The global variable Z is always equal to $2n - 1$.

procedure *Initialize*;
var i : *integer*;
begin
$M := 0$; $E := 0$; $R := -1$; $Z := 2 \times n - 1$;
for $i := 1$ **to** n **do begin**
    $M := M + 1$; $R := R + 1$;
    if $2 \times R = M$ **then begin** $E := E + 1$; $R := 0$ **end**;
    alpha$[i] := i$; *rep*$[i] := i$
    end;
{ Initialize node n as the 0-node }
block$[n] := 1$; *prevBlock*$[1] := 1$; *nextBlock*$[1] := 1$; *weight*$[1] := 0$;
first$[1] := n$; *last*$[1] := n$; *parity*$[1] := 0$; *parent*$[1] := 0$;
{ Initialize available block list }
availBlock $:= 2$;
for $i :=$ *availBlock* **to** $Z - 1$ **do**
    nextBlock$[i] := i + 1$;
nextBlock$[Z] := 0$
end;

The *EncodeAndTransmit* procedure determines the encoding of letter a_j based upon the path from the root of the Huffman tree to a_j's leaf, using the convention that "0" means "to the left," and "1" means "to the right." If a_j has not appeared previously in the message, extra bits are sent to specify which one of the M zero-weight letters has been encountered. These extra bits are computed by the following minimum prefix code: if $1 \le j \le 2R$, then a_j is specified by the $(E+1)$-bit binary representation of $j-1$; otherwise a_j is specified by the E-bit binary representation of $j-R-1$. The system procedure *Transmit* is called for each bit in the encoding to send it to the receiver.

```
procedure EncodeAndTransmit(j : integer);
var i, ii, q, t, root : integer;
begin
q := rep[j]; i := 0;
if q ≤ M then begin   { Encode letter of zero weight }
    q := q − 1;
    if q < 2 × R then t := E + 1 else begin q := q − R; t := E end;
    for ii := 1 to t do begin
        i := i + 1; stack[i] := q mod 2;
        q := q div 2
        end
    q := M;
    end;
if M = n then root := n else root := Z;
while q ≠ root do begin   { Traverse up the tree }
    i := i + 1; stack[i] := (first[block[q]] − q + parity[block[q]]) mod 2;
    q := parent[block[q]] − (first[block[q]] − q + 1 − parity[block[q]]) div 2
    end;
for ii := i downto 1 do Transmit(stack[ii])
end;
```

The *ReceiveAndDecode* function repeatedly calls a system function *Receive* to read one more bit of input until the inputed sequence of 0s and 1s has specified a path to a leaf node in the Huffman tree. Extra bits are read when $k < n - 1$ and a 0-node is reached, in order to determine which zero-weight letter is being transmitted.

```
function ReceiveAndDecode : integer;
var i, q : integer;
begin
if M = n then q := n else q := Z;   { Set q to the root node }
while q > n do   { Traverse down the tree }
    q := FindChild(q, Receive);
if q = M then begin   { Decode 0-node }
    q := 0;
    for i := 1 to E do q := 2 × q + Receive;
    if q < R then q := 2 × q + Receive else q := q + R;
    q := q + 1
    end;
Decode := alpha[q]
end;
```

The function *FindChild* returns the node number of either the left or right child of node j, depending on whether the parity parameter is set to 0 or 1.

```
function FindChild(j, parity : integer) : integer;
var delta, right, gap : integer;
begin
delta := 2 × (first[block[j]] − j) + 1 − parity;
right := rtChild[block[j]]; gap := right − last[block[right]];
if delta ≤ gap then FindChild := right − delta
else begin
    delta := delta − gap − 1;
    right := first[prevBlock[block[right]]]; gap := right − last[block[right]];
    if delta ≤ gap then FindChild := right − delta
    else FindChild := first[prevBlock[block[right]]] − delta + gap + 1
    end
end;
```

The last (inner) **else**-clause is never executed when the Huffman tree is well-formed, such as when *FindChild* is called by *Decode*, but it is needed when *FindChild* is called by *Update* during the modification of the tree.

The procedure *InterchangeLeaves* interchanges the contents of two leaf nodes *e1* and *e2* in the Huffman tree:

```
procedure InterchangeLeaves(e1, e2 : integer);
var temp : integer;
begin
rep[alpha[e1]] := e2; rep[alpha[e2]] := e1;
temp := alpha[e1]; alpha[e1] := alpha[e2]; alpha[e2] := temp
end;
```

The procedure *Update* is the main component of the algorithm. It is called by both *EncodeAndTransmit* and *ReceiveAndDecode* in order to modify the dynamic Huffman tree to take into account the letter just processed.

```
procedure Update(k : integer);
var q, leafToIncrement, bq, b, oldParent, oldParity, nbq, par, bpar : integer;
    slide : boolean;
begin
{ Set q to the node whose weight should increase }
FindNode;
while q > 0 do
    { At this point, q is the first node in its block. Increment q's weight by 1
      and slide q if necessary over the next block to maintain the invariant.
      Then set q to the node one level higher that needs incrementing next }
    SlideAndIncrement;
{ Finish up some special cases involving the 0-node }
if leafToIncrement ≠ 0 then
    begin q := leafToIncrement; SlideAndIncrement end
end;
```

The *FindNode* macro implements the statements appearing before the main loop in the pseudocode for *Update* given at the end of Section 4.

```
macro FindNode ≡
begin
q := rep[k]; leafToIncrement := 0;
if q ≤ M then begin  { A zero weight becomes positive }
    InterchangeLeaves(q, M);
    if R = 0 then begin R := M div 2; if R > 0 then E := E − 1 end;
    M := M − 1; R := R − 1; q := M + 1; bq := block[q];
    if M > 0 then begin
        { Split the 0-node into an internal node with two children.
          The new 0-node is node M; the old 0-node is node M + 1;
          the new parent of nodes M and M + 1 is node M + n }
        block[M] := bq; last[bq] := M; oldParent := parent[bq];
        parent[bq] := M + n; parity[bq] := 1;
        { Create new internal block of zero weight for node M + n }
        b := availBlock; availBlock := nextBlock[availBlock];
        prevBlock[b] := bq; nextBlock[b] := nextBlock[bq];
        prevBlock[nextBlock[bq]] := b; nextBlock[bq] := b;
        parent[b] := oldParent; parity[b] := 0; rtChild[b] := q;
        block[M + n] := b; weight[b] := 0;
        first[b] := M + n; last[b] := M + n;
        leafToIncrement := q; q := M + n
        end
    end
else begin  { Interchange q with the first node in q's block }
    InterchangeLeaves(q, first[block[q]]);
    q := first[block[q]];
    if (q = M + 1) and (M > 0) then
        begin leafToIncrement := q; q := parent[block[q]] end
    end
end;
```

The *SlideAndIncrement* macro increments the weight of node q by 1 and adjusts the tree pointers to reflect the new implicit numbering. Finally, q is set to point to the node one level higher in the tree that needs incrementing next.

```
macro SlideAndIncrement ≡
begin  { q is currently the first node in its block }
bq := block[q]; nbq := nextBlock[bq];
par := parent[bq]; oldParent := par; oldParity := parity[bq];
if ((q ≤ n) and (first[nbq] > n) and (weight[nbq] = weight[bq]))
  or ((q > n) and (first[nbq] ≤ n) and (weight[nbq] = weight[bq] + 1)) then
    begin  { Slide q over the next block }
    slide := true;
    oldParent := parent[nbq]; oldParity := parity[nbq];
    { Adjust child pointers for next-higher level in tree }
    if par > 0 then begin
        bpar := block[par];
        if rtChild[bpar] = q then rtChild[bpar] := last[nbq]
        else if rtChild[bpar] = first[nbq] then rtChild[bpar] := q
        else rtChild[bpar] := rtChild[bpar] + 1;
        if par ≠ Z then
            if block[par + 1] ≠ bpar then
                if rtChild[block[par + 1]] = first[nbq] then
                    rtChild[block[par + 1]] := q
                else if block[rtChild[block[par + 1]]] = nbq then
                    rtChild[block[par + 1]] := rtChild[block[par + 1]] + 1
        end;
    { Adjust parent pointers for block nbq }
    parent[nbq] := parent[nbq] − 1 + parity[nbq]; parity[nbq] := 1 − parity[nbq];
    nbq := nextBlock[nbq];
    end
else slide := false;
if (((q ≤ n) and (first[nbq] ≤ n)) or ((q > n) and (first[nbq] > n)))
  and (weight[nbq] = weight[bq] + 1) then
    begin  { Merge q into the block of weight one higher }
    block[q] := nbq; last[nbq] := q;
    if last[bq] = q then begin  { q's old block disappears }
        nextBlock[prevBlock[bq]] := nextBlock[bq];
        prevBlock[nextBlock[bq]] := prevBlock[bq];
        nextBlock[bq] := availBlock; availBlock := bq
        end
    else begin
        if q > n then rtChild[bq] := FindChild(q − 1, 1);
        if parity[bq] = 0 then parent[bq] := parent[bq] − 1;
        parity[bq] := 1 − parity[bq];
        first[bq] := q − 1
        end
    end
```

```
else if last[bq] = q then begin
    if slide then begin  { q's block is slid forward in the block list }
        prevBlock[nextBlock[bq]] := prevBlock[bq];
        nextBlock[prevBlock[bq]] := nextBlock[bq];
        prevBlock[bq] := prevBlock[nbq]; nextBlock[bq] := nbq;
        prevBlock[nbq] := bq; nextBlock[prevBlock[bq]] := bq;
        parent[bq] := oldParent; parity[bq] := oldParity
        end;
    weight[bq] := weight[bq] + 1
    end
else begin  { A new block is created for q }
    b := availBlock; availBlock := nextBlock[availBlock];
    block[q] := b; first[b] := q; last[b] := q;
    if q > n then begin
        rtChild[b] := rtChild[bq];
        rtChild[bq] := FindChild(q - 1,1);
        if rtChild[b] = q - 1 then parent[bq] := q
        else if parity[bq] = 0 then parent[bq] := parent[bq] - 1
        end
    else if parity[bq] = 0 then parent[bq] := parent[bq] - 1;
    first[bq] := q - 1; parity[bq] := 1 - parity[bq];
    { Insert q's block in its proper place in the block list }
    prevBlock[b] := prevBlock[nbq]; nextBlock[b] := nbq;
    prevBlock[nbq] := b; nextBlock[prevBlock[b]] := b;
    weight[b] := weight[bq] + 1;
    parent[b] := oldParent; parity[b] := oldParity
    end;
{ Move q one level higher in the tree }
if q ≤ n then q := oldParent else q := par
end;
```

The processing in Algorithm Λ is dominated by the calls to *SlideAndIncrement* made by *Update*. Roughly speaking, *SlideAndIncrement* is called once for each level in the tree above the leaf node for the current letter being processed. Each execution of *SlideAndIncrement* involves sliding the current node and then moving up one level in the tree. The floating tree data structure clearly supports these operations in constant time. The net result is that Algorithm Λ does encoding and decoding in real time, that is, the processing time for each letter in the message is proportional to the length of the letter's encoding.

The running time of our implementation is comparable to that of Algorithm FGK. In more than 95% of the time that *SlideAndIncrement* is executed in practice, the node q is neither slid over the next block (that is, we have *slide* = **false**), nor merged into the next block, so the observed execution time is fast. Faster running times can be obtained by replacing the procedures with macros and by breaking up *SlideAndIncrement* into two separate macros—one for leaves and one for internal nodes.

One nice feature of a floating tree, due to the use of implicit numbering, is that

the parent of nodes $2j-1$ and $2j$ is less than the parent of nodes $2j+1$ and $2j+2$ in both the implicit and explicit numberings. Such an invariant is not maintained by the data structure in [Knuth, 1985], for example.

References

D. A. Huffman. "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the Institute of Radio Engineers, 40* (1951), 1098–1101.

D. E. Knuth. "Dynamic Huffman Coding," *Journal of Algorithms, 6* (1985), 163–180.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique